2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware

Jie Chen and Guru Venkataramani
The George Washington University, Washington, DC
{jiec, guruv}@gwu.edu

***Abstract*—As we increasingly rely on computers to process and manage our personal data, safeguarding sensitive information from malicious hackers is a fast growing concern. Among many forms of information leakage, covert timing channels operate by establishing an illegitimate communication channel between two processes and through transmitting information via timing modulation, thereby violating the underlying system's security policy. Recent studies have shown the vulnerability of popular computing environments, such as cloud computing, to these covert timing channels. In this work, we propose a new microarchitecture-level framework, CC-Hunter, that detects the possible presence of covert timing channels on shared hardware. Our experiments demonstrate that CC-Hunter is able to successfully detect different types of covert timing channels at varying bandwidths and message patterns.**

***Keywords*-Covert timing channels; Detection; Shared hardware; Algorithms**

I. Introduction

Information leakage is a fast growing concern affecting computer users that is exacerbated by the increasing amount of shared processor hardware. Every year, there are hundreds of news reports on identity thefts and leaked confidential information to unauthorized parties. NIST National Vulnerability Database reports an increase of 11× in the number of information leak/disclosure-related software issues over the past five years (2008-2013), compared to the prior decade (1997-2007) [1].

Covert timing channels are information leakage channels where a trojan process intentionally modulates the timing of events on certain shared system resources to illegitimately reveal sensitive information to a spy process. Note that the trojan and the spy do not communicate explicitly through send/receive or shared memory, but covertly via modulating certain events (Figure 1). In contrast to side channels where a process unintentionally leaks information to a spy process, covert timing channels have an insider trojan process (with higher privileges) that intentionally colludes with a spy process (with lower privileges) to exfiltrate the system secrets.

To achieve covert timing based communication on shared processor hardware, a fundamental strategy used by the trojan process is modulating the timing of events through intentionally creating conflicts[1]*. The spy process deciphers the secrets by observing the differences in resource access times*. On hardware units such as compute logic and wires (buses/interconnects), the trojan creates conflicts by introducing distinguishable *contention* patterns on the shared resource. On caches, memory and disks, the trojan creates conflicts by *intentionally replacing* certain memory blocks such that the spy can decipher the message bits based on the memory hit/miss latencies. This basic strategy of creating conflicts for timing modulation has been observed in numerous covert timing channel implementations [2], [3], [4], [5], [6], [7], [8], [9], [10].

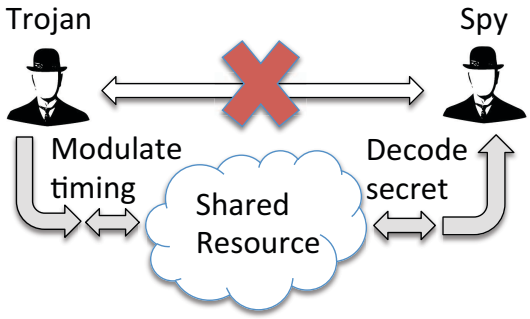


Figure 1: A Covert Timing Channel using timing modulation on a shared resource to divulge secrets

In this paper, we propose *CC-Hunter*, a novel framework that *detects* the presence of covert timing channels by dynamically tracking conflict patterns on *shared processor hardware*. CC-Hunter offers low-cost hardware support that gathers data on certain key indicator events during program execution, and provides software support to compute the likelihood of covert timing channels on a specific shared hardware. Many prior works on covert channels have studied mitigation techniques for *specific* hardware resources such as caches [7] and bus/interconnect [2], [3], [11]. These techniques can neatly complement our CC-Hunter framework by mitigating the damages caused by covert timing channels after detection. Note that detecting network-based covert information transfer channels [12], [13], software-based channels (e.g., data objects, file locks) [14] and side channels [15], [16] are beyond the scope of our work.

Our framework can be extremely beneficial to users as we transition to an era of running our applications on remote servers that host programs from many different users. Recent studies [6], [9] show how popular computing environments

[1]We use "conflict" to collectively denote methods that alter either the latency of a single event or the inter-event intervals.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.42

like cloud computing are vulnerable to covert timing channels. Static techniques to eliminate timing channel attacks such as program code analyses are virtually impractical to enforce on every third-party software, especially when most of these applications are available only as binaries. Also, adopting strict system usage policies (such as minimizing system-wide resource sharing or fuzzing the system clock to reduce the possibility of covert timing channels) could adversely affect the overall system performance. To overcome these issues, CC-Hunter's dynamic detection is a desirable first step before adopting damage control strategies like limiting resource sharing or bandwidth reduction.

In summary, the contributions of our paper are:

1) We propose *CC-Hunter*, a novel microarchitecture-level framework to detect shared hardware-based covert timing channels by monitoring for *conflicts*.

2) We design algorithms that extract recurrent (yet, sometimes irregular) conflict patterns used in covert transmission, and show our implementation in hardware and software.

3) We evaluate the efficacy of our solution using covert timing channels on three different types of shared hardware resources, namely wires (memory bus/QPI), logic (integer divider) and memory (shared L2 cache). Our experiments demonstrate that CC-Hunter is able to successfully detect different types of covert timing channels at varying bandwidths and message patterns, and has zero false alarms for the cases we tested.

II. Understanding Covert Timing Channels

Trusted Computer System Evaluation Criteria (or TCSEC, The Orange Book) [17] defines a covert channel as *any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy*. In particular, covert timing channels are those that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information.

Note that, between the trojan and the spy, the task of constructing a reliable covert channel is not very simple. Covert timing channels implemented on real systems take significant amounts of synchronization, confirmation and transmission time even for relatively short-length messages. As examples, (1) Okamura et al. [4] construct a memory load-based covert channel on a real system, and show that it takes 131.5 seconds just to covertly communicate 64 bits in a reliable manner achieving a bandwidth rate of 0.49 bits per second; (2) Ristenpart et al. [6] demonstrate a memory-based covert channel that achieves a bandwidth of 0.2 bits per second. This shows that the covert channels create non-negligible amounts of traffic on shared resources to accomplish their intended tasks.

TCSEC points out that a covert channel bandwidth exceeding a rate of one hundred (100) bits per second is classified as a high bandwidth channel based on the observed data transfer rates between several kinds of computer systems. In any computer system, there are a number of relatively low-bandwidth covert channels whose existence is deeply ingrained in the system design. If bandwidth-reduction strategy to prevent covert timing channels were to be applied to all of them, it becomes an impractical task. Therefore, TCSEC points out that channels with maximum bandwidths of less than 0.1 bit per second are generally not considered to be very feasible covert timing channels. This does not mean that it is impossible to construct very low bandwidth covert timing channel, just that it becomes very expensive and difficult for the adversary (spy) to extract any meaningful information out of the system.

III. Threat Model and Assumptions

Our threat model assumes that the trojan wants to *intentionally* communicate the secret information to the spy covertly by modulating the timing on certain hardware. We assume that the spy is able to seek the services of a compromised trojan that has sufficient privileges to run inside the target system. As confinement mechanisms in software improve, hardware-based covert timing channels will become more important. So, we limit the scope of our work to shared processor hardware.

A hardware-based covert timing channel could have noise due to two factors- (1) processes other than the trojan/spy using the shared resource *frequently*, (2) the trojan artificially inflating the patterns of random conflicts to evade detection by CC-Hunter. In both cases, the reliability of covert communication is severely affected resulting in loss of data for the spy as evidenced by many prior studies [10], [18], [19]. For example, on a cache-based covert timing channel, Xu et al. [10] find that the covert transmission error rate is at least 20% when 64 concurrent users share the same processor with the trojan/spy. Therefore, we point out that it is impossible for a covert timing channel to just randomly inflate conflict events or operate in noisy environments simply to evade detection. In light of these prior findings, we model moderate amounts of interference by running a few other (at least three) active processes alongside the trojan/spy processes in our experiments.

In this work, our focus is on the detection of covert timing channels rather than showing how to actually construct or prevent them. We do not evaluate the robustness of covert communication itself that has been demonstrated adequately by prior work [6], [9], [10].

We assume that covert timing based communication happens through recurrent patterns of conflicts over non-trivial intervals of time. CC-Hunter cannot detect the covert timing attacks that happen instantly where the spy has the ability to gain sensitive information in one pass. Also, covert timing channels that employ sophisticated combinations of timing and storage channels at both hardware and software layers

are not considered in this work. Finally, we assume that the system software modules (including the operating system kernel and security enforcing layers) are trusted.

IV. Design Overview

From the perspective of covert timing channels that exploit shared hardware, there are two categories–

(1) Combinational structures such as logic and wires, relying on patterns of high and low contention to communicate on the corresponding shared resource. Consequently, a recurrent (yet sometimes irregular) pattern of contention (conflicts) would be observed in the corresponding event time series during covert communication.

(2) Memory structures, such as caches, DRAM and disks, using intentional replacement of memory blocks (previously owned by the spy) to create misses. As a result, we observe a recurrent pattern of cache conflict misses.

We design algorithms to identify the recurrent patterns in the corresponding event time series[2]. Our algorithms look for patterns of conflicts, a fundamental property of covert timing channels. Hence, even if the trojan and spy dynamically change their communication protocol, CC-Hunter should still be able to detect them based on conflict patterns.

To demonstrate our framework's effectiveness, we use three realistic covert timing channel implementations, two of which (shared caches [10], memory bus [9]) have been demonstrated successfully on Amazon EC2 cloud servers. We evaluate using a full system environment by booting MARSSx86 [21] with Ubuntu 11.04. The simulator models a quad-core processor running at 2.5 GHz, each core with two hyperthreads, and has a few (at least three) other active processes to create real system interference effects. We model a private 32 KB L1 and 256 KB L2 caches. Prior to conducting our experiments, we validated the timing behavior of our covert channel implementations running on MARSSx86 against the timing measurements in a real system environment (dual-socket Dell T7500 server with Intel 4-core Xeon E5540 processors at 2.5 GHz, Ubuntu 11.04).

Note that the three covert timing channels described below are randomly picked to test our detection framework. CC-Hunter is neither limited to nor derived from their specific implementations, and can be used to detect covert timing channels on all shared processor hardware using recurrent patterns of conflicts for covert communication.

A. Covert Timing Channels on Combinational Hardware

To illustrate the covert timing channels that occur on combinational structures and their associated indicator events, we choose the memory bus and integer divider unit (Wang et al [7] showed a similar implementation using multipliers).

[2]Our solution is inspired from studies in neuroscience that analyze patterns of neuronal activity to understand the physiological mechanisms associated with behavioral changes [20].

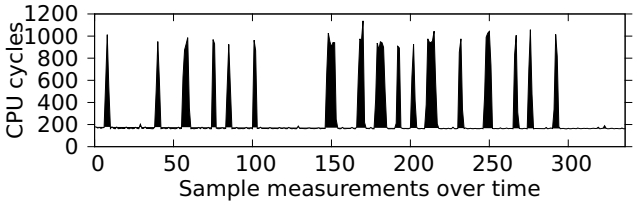


Figure 2: Average latency per memory access (in CPU cycles) in Memory Bus Covert Channel

In the case of the memory bus covert channel, when the trojan wants to transmit a '1' to the spy, it intentionally performs an atomic unaligned memory access spanning two cache lines. This action triggers a memory bus lock in the system, and puts the memory bus in contended state for most modern generations of processors including Intel Nehalem and AMD K10 family. The trojan repeats the atomic unaligned memory access pattern for a number of times to sufficiently alter the memory bus access timing for the spy to take note of the '1' value transmission. Even on x86 platforms that have recently replaced the shared memory bus with QuickPath Interconnect (QPI), the bus locking behavior is still emulated for atomic unaligned memory transactions spanning multiple cache lines [22]. Consequently, delayed interconnect access is still observable in QPI-based architectures. To communicate a '0', the trojan simply puts the memory bus in un-contended state. The spy deciphers the transmitted bits by accessing the memory bus intentionally through creating cache misses. It times its memory accesses and detects the memory bus contention state by measuring the average latency. The spy accumulates a number of memory latency samples to infer the transmitted bit. Figure 2 shows the average loop execution time observed by the spy for a randomly-chosen 64-bit credit card number. A contended bus increases the memory latency enabling the spy to infer '1', and an un-contended bus to infer '0'.

For the integer division unit, both the trojan and the spy processes are run on the same core as hyperthreads. The trojan communicates '1' by creating a contention on all of the division units by executing a fixed number of instructions. To transmit a '0', the trojan puts all of the division units in an un-contended state by simply executing an empty loop. The spy covertly listens to the transmission by executing loop iterations with a constant number of integer division operations and timing them. A '1' is inferred on the spy side using iterations that take longer amounts of time (due to contentions on the divider unit created by the trojan), and '0' is inferred when the iterations consume shorter time. Figure 3 shows the average latency per loop iteration as observed by the spy for the same 64-bit credit card number chosen for memory bus covert channel. We observe that the loop latency is high for '1' transmission and remains low for '0' transmission.

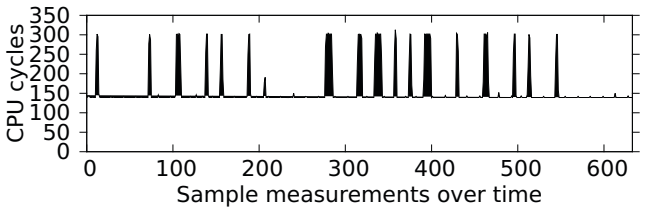
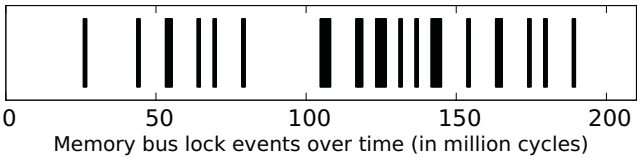
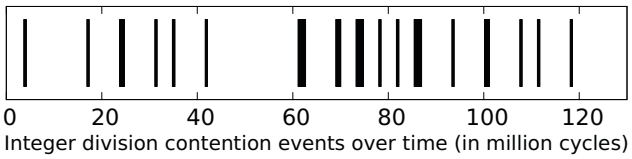


Figure 3: Average loop execution time (in CPU cycles) in Integer Divider Covert Channel



(a) Memory Bus



(b) Integer Divider

Figure 4: Event Train plots for Memory Bus and Integer Divider showing burst patterns

B. Recurrent Burst Pattern Detection

The *first* step in detecting covert timing channels is to identify the event that is behind the hardware resource contention. In the case of the memory bus covert channel, the event to be monitored is the memory bus lock operation. In the case of the integer division covert channel, the event to be monitored is the number of times a division instruction from one process (hardware context) waits on a *busy* divider occupied by an instruction from another process (context). Note that not all division operations fall in this category.

The *second* step is to create an *Event Train*, i.e., a uni-dimensional time series showing the occurrence of events (Figures 4a and 4b). We notice a large number of thick bands (or bursty patterns of events) whenever the trojan intends to covertly communicate a '1'.

As the *third* step, we analyze the event train using our recurrent burst pattern detection algorithm. This step consists of two parts: (1) check whether the the event train has significant contention clusters (bursts), and (2) determine if the time series pattern exhibits recurrent patterns of bursts.

Our algorithm is as follows:

1) *Determine the interval (Δt) for a given event train to calculate event density*. Δt is the product of the inverse of average event rate and α, an empirical constant determined using the maximum and minimum achievable covert timing channel bandwidth rates on a given shared hardware. In simple terms, Δt is the observation window to count the number of event occurrences within that interval. The value of Δt can be picked from a wide range, and is tempered by the α factor which ensures that Δt is neither too low (when the probability of a certain number of events within Δt follows Poisson distribution) nor too high (when the probability of a certain number of events within Δt follows normal distribution). For covert timing channel with the memory bus, Δt is determined as 100,000 CPU cycles (or 40 μs), and for the covert timing channel using integer

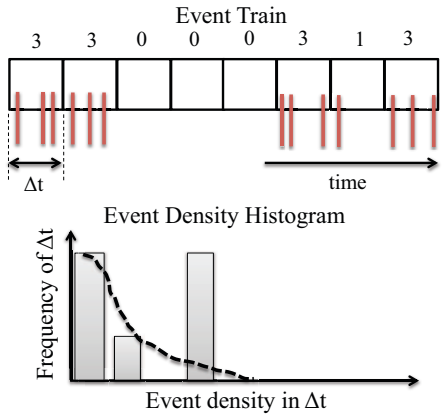
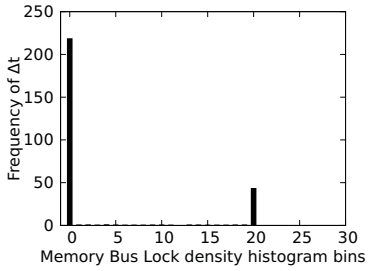
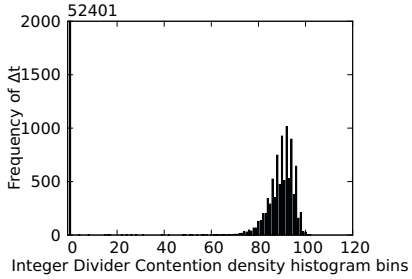


Figure 5: Illustration of Event Train and its corresponding Event Density Histogram. The distribution is compared against the Poisson Distribution shown by the dotted line to detect the presence of burst patterns.



(a) Memory Bus



(b) Integer Division Unit

Figure 6: Event Density Histograms for Covert Timing Channels using Memory Bus and Integer Divider.

divisions, Δt is determined as 500 CPU cycles (or 200 ns).

2) *Construct the event density histogram using Δt.* For each interval of Δt, the number of events are computed, and an event density histogram is constructed to subsequently estimate the probability distribution of event density. An illustration is shown in Figure 5. The x-axis in the histogram plot shows the range of Δt bins that have a certain number of events. Low density bins are to the left, and as we move right, we see the bins with higher numbers of events. The y-axis shows the number of Δt's within each bin.

3) *Detect burst patterns*. From left to right in the histogram, threshold density is the first bin which is smaller than the preceding bin, and equal or smaller than the next bin. If there is no such bin, then the bin at which the slope of the fitted curve becomes gentle is considered as the threshold density. If the event train has burst patterns, there will be two distinct distributions- (1) one where the mean is below 1.0 showing the non-bursty periods, and (2) one where the mean is above 1.0 showing the bursty periods present in the right tail of the event density histogram. Figure 6 shows the event density histogram distributions for covert timing channels involving bursty contention patterns on the memory bus and the integer division unit. For both timing channels, we see significant non-burst patterns in the histogram bin#0. In the case of the memory bus channel, we see significant bursty pattern at histogram bin#20. For the integer division channel, we see a very prominent second distribution (burst pattern) between bins#84 and #105 with peak around bin#96.

4) *Identify significant burst patterns (contention clusters) and filter noise*. To estimate the significance of burst distribution and filter random (noise) distributions, we compute the likelihood ratio[3] of the second distribution. Empirically, based on observing realistic covert timing channels [11], [9], we find that the likelihood ratio of the burst pattern distribution tends to be at least 0.9 (even on very low bandwidth covert channels such as 0.1 bps). On the flip-side, we observe this likelihood ratio to be less than 0.5 among regular programs that have no known covert timing channels despite having some bursty access patterns. We set a conservative threshold for likelihood ratio at 0.5, i.e., all event density histograms with likelihood ratios above 0.5 are considered for further analysis.

5) *Determine the recurrence of burst patterns.* Once the presence of significant burst patterns are identified in the event series, the next step is to check for recurrent patterns of bursts. We limit the *window of observation* to 512 OS time quanta (or 51.2 secs, assuming a time quantum of 0.1 secs), to avoid diluting the significance of event density histograms involved in covert timing channels. We develop a pattern clustering algorithm that performs two basic steps- (1) discretize the event density histograms into strings, and (2) use k-means clustering to aggregate similar strings. By analyzing the clusters that represent event density histograms with significant bursts, we can find the extent to which burst patterns recur, and hence detect the possible presence of a covert timing channel. Since we use clustering to extract recurring burst patterns, our algorithm can detect covert timing channels *regardless* of burst intervals (i.e., even on low-bandwidth irregular bursts or in the presence of random noise due to interference from the system environment).

[3]Likelihood ratio is defined as the number of samples in the identified distribution divided by the total number of samples in the population [23]. We omit bin#0 from this computation since it does not contribute to any contention.

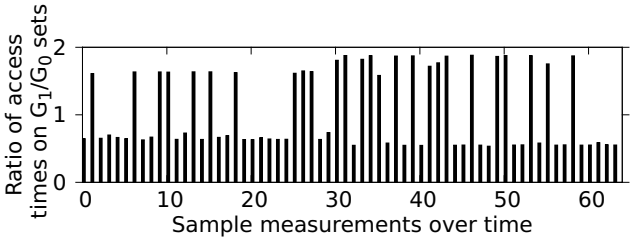


Figure 7: Ratios of cache access times between G_1 and G_0 cache sets in Cache Covert Channel

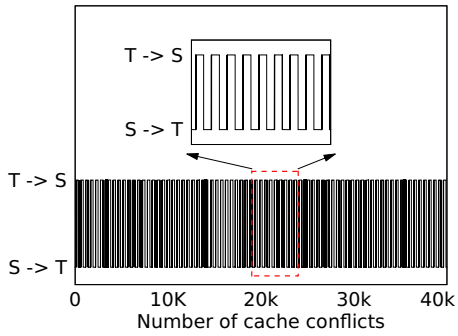
C. Covert Timing Channel on Shared Cache

We use the L2 cache-based timing channel demonstrated by Xu et al [10]. To transmit a '1', the trojan visits a dynamically[4] determined group of cache sets (G_1) and replaces all of the constituent cache blocks, and for a '0' it visits another dynamically determined group of cache sets (G_0) and replaces all of the constituent cache blocks. The spy infers the transmitted bits as follows: It replaces all of the cache blocks in G_1 and G_0, and times the accesses to the G_1 and G_0 sets separately. If the accesses to G_1 sets take longer than the G_0 sets (that is, all of the G_1 sets resulted in cache misses and G_0 sets were cache hits), then the spy infers '1'. Otherwise, if the accesses to G_0 sets take longer than the G_1 sets (that is, all of the G_0 sets resulted in cache misses and G_1 sets were cache hits), then the spy infers a '0'. Figure 7 shows the ratio of the average cache access latencies between G_1 and G_0 cache set blocks observed by the spy for the same 64-bit randomly generated credit card number. A '1' is inferred for ratios greater than 1 (i.e., G_1 set access latencies are higher than G_0 set access latencies) and a '0' is inferred for ratios less than 1 (i.e., G_1 set access latencies are lower than G_0 set access latencies).

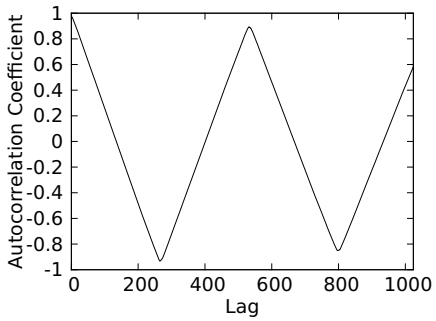
D. Oscillatory Pattern Detection

Unlike combinational structures where timing modulation is performed by varying the inter-event intervals (observed as bursts and non-bursts), cache based covert timing channels rely on the latency of events to perform timing modulation. To transmit a '1' or a '0', the trojan and the spy create

[4]The cache sets, where conflict misses are created and detected for covert transmission, are chosen during the covert channel synchronization phase.



(a) Event Train (T→S: Trojan's(S) conflict misses with Spy(S) and S→T: S's conflict misses with T)



(b) Autocorrelogram

Figure 8: Oscillatory pattern of L2 cache conflict misses between trojan and spy. An autocorrelogram is shown for the conflict miss event train.

a sufficient number of conflict events (cache misses) alternatively among each other that lets the spy decipher the transmitted bit based on the average memory access times (hit/miss). Note that this leads to oscillatory patterns of conflicts between the trojan and spy contexts.

Oscillation is defined as a property of periodicity in an event train. This is different from bursts that are specific periods of high frequency event occurrences in the event train. Oscillation of an event train is detected by measuring its autocorrelation [24]. Autocorrelation is the correlation coefficient of the signal with a time-lagged version of itself, i.e., the correlation coefficient between two values of the same variable, X_i and X_{i+p} separated by a lag p.

In general, given the measurements of a variable X, (X_1, X_2, ..., X_N) at time instances of t (t_1, t_2, ..., t_N), the autocorrelation coefficient r_p at a time lag of p and mean of $\bar{X}$ is defined as,

$$r_p = \frac{\sum_{i=1}^{n-p}(X_i - \bar{X}).(X_{i+p} - \bar{X})}{\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

The autocorrelation function is primarily used for two purposes– (1) detecting non-randomness in data, (2) identifying an appropriate time series model if the data values are not random [24]. To satisfy #1, computing the autocorrelation coefficient for a lag value of 1 (r_1) is sufficient. To satisfy #2, autocorrelation coefficients for a sequence of lag values should exhibit significant periodicity.

An autocorrelogram is a chart showing the autocorrelation coefficient values for a sequence of lag values. An oscillation pattern is inferred when the autocorrelation coefficient shows significant periodicity with peaks sufficiently high for certain lag values (i.e., the values of X correlates highly with itself at lag distances of k_1, k_2 etc.).

Figure 8 shows the oscillation detection method for the covert timing channel on shared cache. In particular, Figure 8a shows the event train (cache conflict misses) annotated by whether the conflicts happen due to the trojan replacing the spy's cache sets, or vice versa. "T→S" denotes the Trojan (T) replacing the Spy's(S) blocks because the spy had previously displaced those same blocks owned by the trojan at that time. Since the conflict miss train shows a dense cluttered pattern, we show a legible version of this event train as an inset of Figure 8a.

The conflict misses that are observed within each observation window (typically one OS time quantum) are used to construct a conflict miss event train plot. Every conflict miss in the event train is denoted by an identifier based on the replacer and the victim contexts. Note that every ordered pair of trojan/spy contexts have unique identifiers. For example "S→T" is assigned '0' and "T→S" is assigned "1". The autocorrelation function is computed on this conflict miss event train. Figure 8b shows the autocorrelogram of the event train. A total of 512 cache sets were used in G_1 and G_0 for transmission of "1" or "0" bit values. We observe that, at a lag value of 533 (that is very close to the actual number of conflicting sets in the shared cache, 512), the autocorrelation value is highest at about 0.893. The slight offset from the actual number of conflicting sets was observed due to random conflict misses in the surrounding code and the interference from conflict misses due to other active contexts sharing the cache. At a lag value of 512, the autocorrelation coefficient value was also high (≈0.85). To evade detection, the trojan/spy may (with some effort) may deliberately introduce noise through creating cache conflicts with other contexts. This may potentially lower autocorrelation coefficients, but we note that the trojan/spy may face a much bigger problem in reliable transmission due to higher variability in cache access latencies.

V. Implementation

In this section, we show the hardware modifications and software support to implement our CC-Hunter framework.

A. Hardware Support

In current microprocessor architectures, we note that most hardware units are shared by multiple threads, especially with the widespread adoption of Simultaneous Multi-

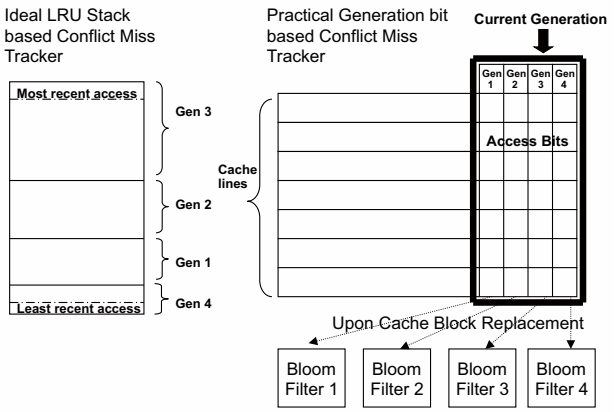


Figure 9: Conflict Miss Tracker implementation

Threading (SMT). Therefore, all of the microarchitectural units are potential candidates for timing channel mediums.

The Instruction Set is augmented with a special instruction that lets the user program the CC-auditor and choose certain hardware units to audit. This special instruction is a privileged instruction that only a subset of system users (usually the system administrator) can utilize for system monitoring. The hardware units have a monitor bit, which when set by the CC-auditor, places the hardware unit under audit for covert timing channels. The hardware units are wired to fire a signal to the CC-auditor on the occurrence of certain key indicator events seen in covert timing channels.

In super-secure environments, where performance constraints can be ignored, CC-auditor hardware can be enabled to monitor all shared hardware structures. However, this would incur unacceptable performance overheads in most real system environments. Therefore, to minimize CC-Hunter implementation complexity, we design CC-auditor with the capability to monitor up to *two* different hardware units *at any given time*. The user (system administrator) is responsible for choosing the two shared hardware units to monitor based on his knowledge of the current system jobs. We believe that this hardware design tradeoff can prevent unnecessary overheads on most regular user applications.

For most of the core components like execution clusters and logic, the indicator events are conflicts detected by a hardware context when another context is already using them. On certain uncore components like the memory bus, conflicts are created using special events such as bus locks.

To accumulate the event signals arriving from the hardware units, the CC-Auditor contains (1) two 32-bit count-down registers initialized to the computed values of Δt based on the two microarchitecture units under monitor (Section IV-B), (2) two 16-bit registers to accumulate the number of event occurrences within Δt, and (3) two histogram buffers with 128 entries (each entry is 16 bits long) to record the event density histograms. Whenever the event signal arrives from the unit under audit, the accumulator register is incremented by one. At the end of each Δt, the corresponding 16-bit accumulator value is updated against its entry in the histogram buffer, and the count-down register is reset to Δt. At the end of OS time quantum, the histogram buffers are recorded by the software module.

For memory structures such as caches, conflict misses are utilized for covert data transmission. A conflict miss happens in a set associative cache when several blocks map into the same cache set and replace each other even when there is enough capacity left in the cache. When the number of blocks in a set exceeds the cache associativity, a block, A, will be evicted even though better candidates for eviction may exist in the cache in other sets. If A is accessed again before those better candidates are replaced, that access is a conflict miss. Note that a fully associative cache of the same capacity would have kept A in the cache, and not incur a conflict miss (due to full associativity). Therefore, to accurately identify the conflict misses in a set-associative cache, we need to check whether the (incoming) block would be retained (not be prematurely replaced) in a fully-associative cache. Ideally, to do so, we need a fully-associative stack with LRU (Least Recently Used) replacement policy that tracks the access recency information for cache blocks. This ideal scheme is expensive due to the frequent stack updates necessary for every cache block access.

Figure 9 shows our practical implementation that approximates the LRU stack access-recency information [25]. Our scheme maintains four generations that are ordered by age. Each generation consists of a set of blocks, and all the blocks in a younger generation have been accessed more recently than any block in an older generation. This means that the blocks in the youngest generation are the blocks that would be at the top of the LRU stack, the next (older) generation corresponds to the next group on the LRU stack, etc. Note that the blocks within a generation itself are unordered. A new empty generation is started when the number of recently accessed cache blocks reaches a threshold, T (that equals to $\#totalcacheblocks(N)/4$ and roughly corresponds to reaching 25% capacity in an ideal LRU stack).

To implement our conflict miss tracker, each cache block metadata field is augmented with four bits to record the generations during which the block was accessed, and three more bits are added to track the current owner context (assuming four cores with two SMT threads). The youngest generation bit in the cache block metadata is set upon block access (to emulate the placement of a cache block at the top of the LRU stack). During block replacement, the replaced cache tags are recorded in a compact three-hash bloom filter corresponding to the latest generation when the block was accessed (to remember its premature removal from the cache before reaching full capacity). If the incoming cache tag is

Table I: Area, Power and Latency Estimates of CC-Auditor

	Histogram Buffers	Registers	Conflict Miss Detector
Area(mm^2)	0.0028	0.0011	0.004
Power(mW)	2.8	0.8	5.4
Latency(ns)	0.17	0.17	0.12

found in one of the bloom filters[5], it denotes a conflict miss since the (incoming) block was removed recently from the cache prior to reaching the full N-block cache capacity.

When the number of accessed blocks reaches the threshold, T, the oldest generation is discarded by flash clearing the corresponding generation column in the cache metadata and all of the bits in the respective bloom filter. This action represents the removal of entries from the bottom of the LRU stack. The generation tracking bits are reused similar to circular buffers (Figure 9), and a separate two bit register (per cache) holds the ID of the current youngest generation.

Since our scheme tracks the conflict misses on all of the cache blocks, we can *accurately identify* the conflict miss event patterns *even if arbitrary cache sets* were used by the trojan/spy for covert communication. Inside our CC-auditor, we maintain two alternating 128-byte vector registers that, *upon every conflict miss identified by our practical conflict miss tracker*, records the three-bit context IDs of the replacer (context requesting the cache block) and the victim (current owner context in the cache block metadata). When one vector register is full, the other vector register begins to record the data. Meanwhile, the software module records the vector contents in the background (to prevent process stalling), and then clears the vector register for future use. Such tracking of the replacer and the victim represents the construction of conflict miss event train. An autocorrelogram on the conflict miss event series can help detect the presence of cache conflict-based covert timing channel (Section IV-D). Oscillation detection method (Section IV-C) uses this practical implementation to identify cache conflict misses. Occasionally, during context switches, the trojan or spy may be scheduled to different cores. Fortunately, the OS (and software layers) have the ability to track the possible migration of processes during context switches. With such added software support, we can identify trojan/spy pairs correctly despite their migration.

1) Area, Latency and Power Estimates of CC-auditor: We use Cacti 5.3 [26] to estimate the area, latency and power needed for our CC-auditor hardware. Table I shows the results of our experiments. For the two histogram buffers, we model 128-entries that are each 16-bits long. For registers, we model two 128-byte vector registers, two 16-bit accumulators, and two 4-byte countdown registers. For the conflict miss detector, we model 4 three-hash bloom filters with $(4 \times \#totalcacheblocks)$ bits, seven metadata bits per cache block (four generation bits plus three bits of owner context). Our experimental results show that the CC-Hunter hardware area overheads are insignificant compared to the total chip area (e.g., 263 mm^2 for Intel i7 processors [27]). The CC-auditor hardware has latencies that are less than the processor clock cycle time (0.33 ns for 3 GHz). Also, the extra bits in the cache metadata array increase the cache access latency slightly by about 1.5%, and is unlikely to extend the clock cycle period. Similarly, the dynamic power drawn by CC-auditor hardware is in the order of a few milliwatts compared to 130 W peak in Intel i7 [27].

[5]A hit in one of the bloom filters means that the cache block was accessed in the corresponding generation, but was replaced to accommodate another more recently accessed block in the same or one of the younger generations.

B. Software Support

In order to place a microarchitectural unit under audit, the user requests the CC-auditor through a special software API exported by the OS, where the OS performs user authorization checks. This is to prevent the sensitive system activity information from being exploited by attackers.

A separate daemon process (part of CC-Hunter software support) accumulates the data points by recording the histogram buffer contents at each OS time quantum (for contention-based channels) or the 128-byte vector register (for oscillation-based channels). Lightweight code is carefully added to avoid perturbing the system state, and to record performance counters as accurately as possible [28]. To further reduce perturbation effects, the OS schedules the CC-Hunter monitors on (currently) un-audited cores.

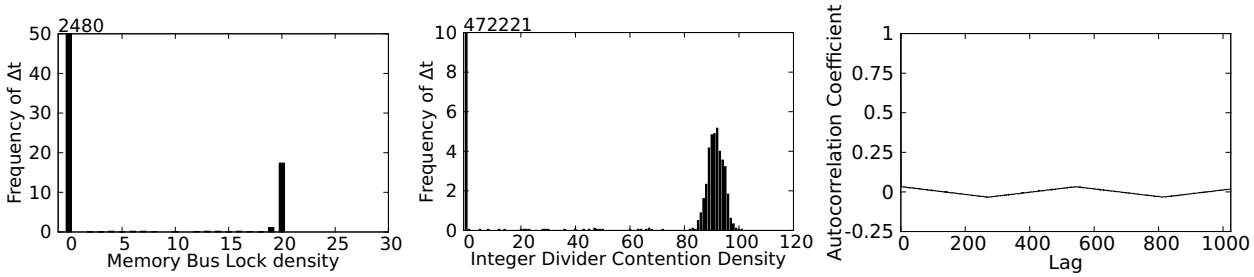
Since our analysis algorithms are run as background processes, they incur minimal effect on system performance. Our pattern clustering algorithm is invoked every 51.2 secs (Section IV-B) and takes 0.25 secs (worst case) per computation. We note that further optimizations such as feature dimension reduction improves the clustering computation time to 0.02 secs (worst case). Our autocorrelation analysis is invoked at the end of every OS time quantum (0.1 secs) and takes 0.001 secs (worst case) per computation.

VI. Evaluation and Sensitivity Study

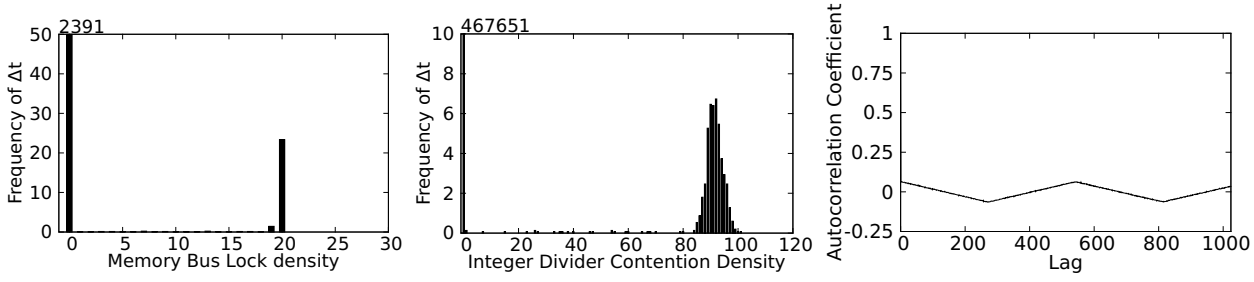
A. Varying Bandwidth Rates

We conduct experiments by altering the bandwidth rates of three different covert timing channels from 0.1 bps to 1000 bps. The results (observed over a window of OS time quantum, 0.1 secs) are shown in Figure 10. While the magnitudes of Δt frequencies decrease for lower bandwidth contention-based channels, the likelihood ratios for second (burst) distribution are still significant (higher than 0.9)[6] On low-bandwidth cache covert channels such as 0.1 bps, despite observing periodicity in autocorrelation values, we note that their magnitudes do not show significant strength.

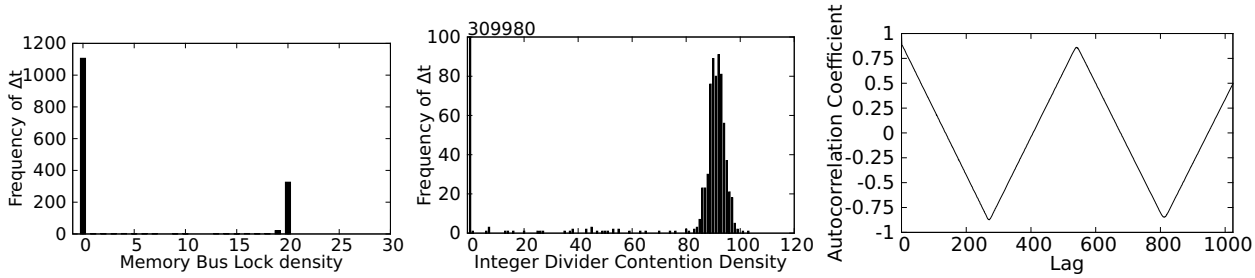
[6]The histogram bins for the second distribution (covert transmission) are determined by the number of successive conflicts needed to reliably transmit a bit and the timing characteristics of the specific hardware resource.



(a) Bandwidth=0.1bps

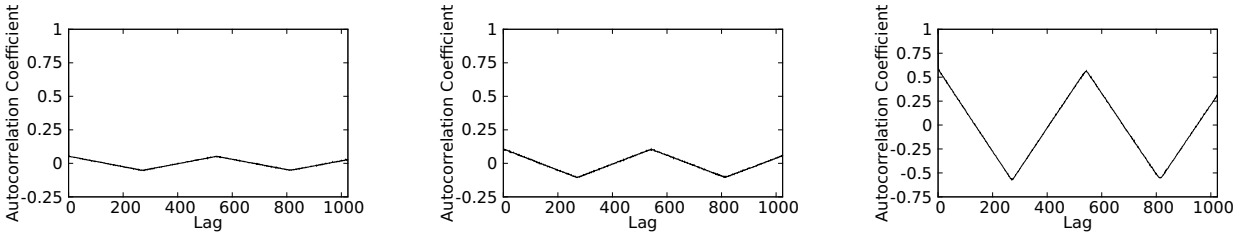


(b) Bandwidth=10bps



(c) Bandwidth=1000bps

Figure 10: Bandwidth test using Memory Bus, Integer Divider and Cache Covert Channels



(a) 0.75X OS time quantum (b) 0.5X OS time quantum (c) 0.25X OS time quantum

Figure 11: Autocorrelograms for 0.1 bps cache covert channels at reduced observation window sizes

We conduct additional experiments by decreasing the windows of observation to fractions of OS time quantum on 0.1 bps channel. This fine grain analysis is especially useful for lower-bandwidth channels that create a certain number of conflicts per second (needed to reliably transmit a bit) frequently followed by longer periods of dormancy. Figure 11 shows that, as we reduce the sizes of the observation window, the autocorrelograms show significant repetitive peaks for 0.1 bps channel. Our experiments suggest that autocorrelation analysis at finer granularity observation windows can detect lower-bandwidth channels more effectively.

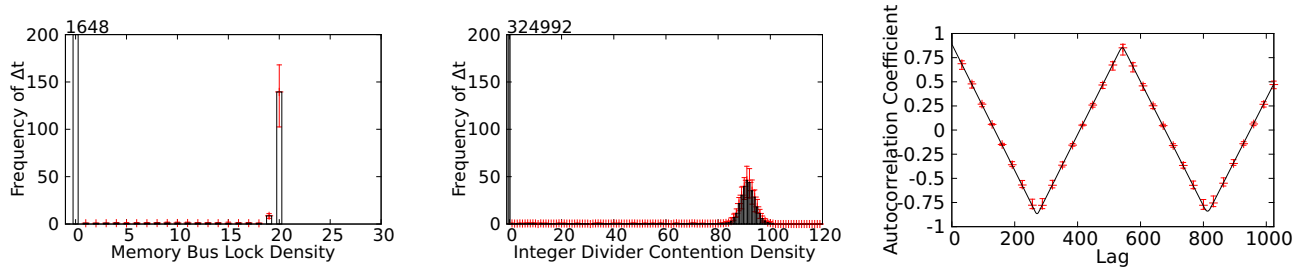
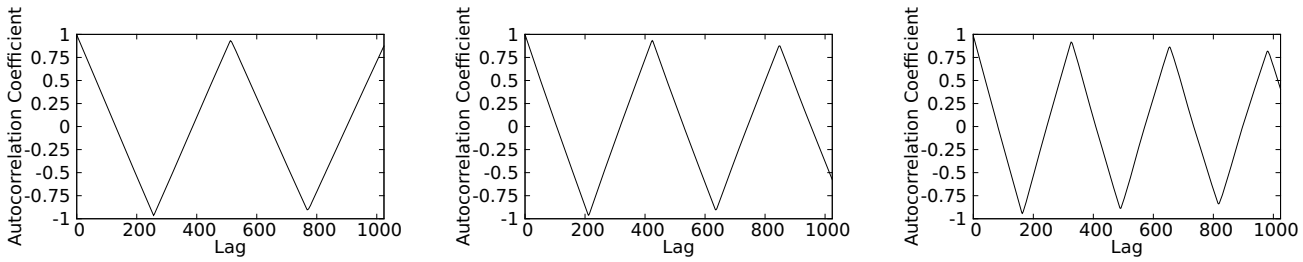


Figure 12: Test with 256 randomly generated 64-bit messages on Memory Bus, Integer Divider and Cache covert channels. Black (thick) bars are the means, and the red (annotations) arrows above them show the range (min, max).



(a) #sets for covert channel=256 (b) #sets for covert channel=128 (c) #sets for covert channel=64

Figure 13: Autocorrelograms for cache covert channel with varying numbers of cache sets for communication

B. Encoded Message patterns

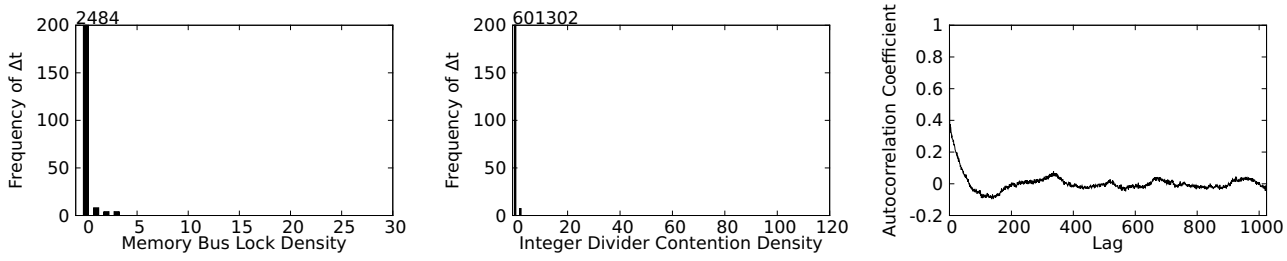
To simulate encoded message patterns that the trojan may use to transmit messages, we generate 256 random 64-bit combinations, and use them as inputs to the covert timing channels. Our experimental results are shown in Figure 12. Mean values of histogram bins are shown by dark bars that are annotated by the range (maximum, minimum) of bin values observed across the 128 runs. Despite variations in peak magnitudes of Δt frequencies (especially in integer divider), we notice that our algorithm still shows significant second distributions with likelihood ratios above 0.9. For autocorrelograms in cache covert channels, we notice insignificant deviations in autocorrelation coefficients.

C. Varying Cache Channel Implementations

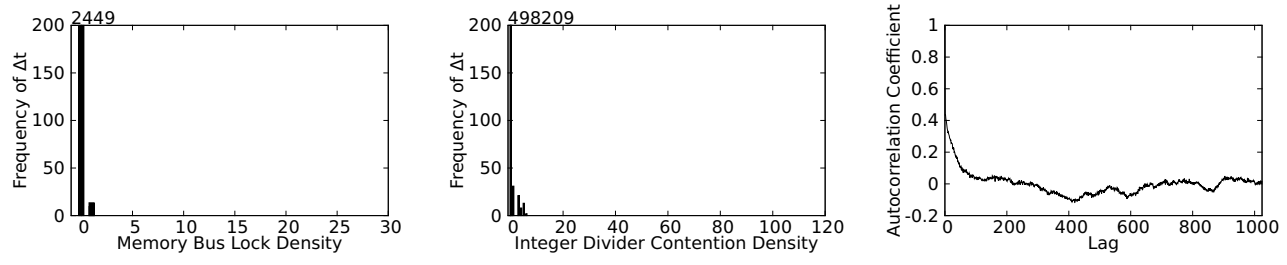
We implement the cache covert timing channels by varying the number of cache sets used for bit transmission from 64 to 512. In Figure 13, we find that the autocorrelograms in all of the cases show significant periodicity in autocorrelation with maximum peak correlation values of around 0.95, a key characteristic observed in covert timing channels. For covert channels that uses a smaller number of cache sets, note that random conflict misses occurring on other cache sets and interference from other active processes increase the wavelength of the autocorrelogram curve beyond the expected values (typically the number of cache sets used in covert communication).

D. Testing for False Alarms

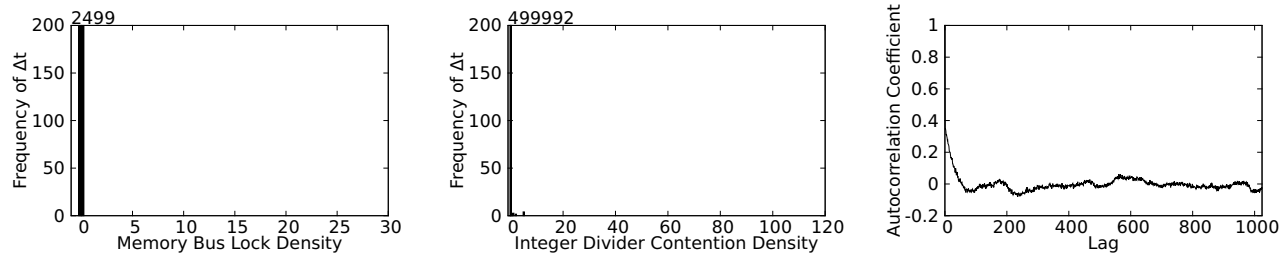
We test our recurrent burst and oscillation pattern algorithms on 128 pair-wise combinations of several standard SPEC2006, Stream and Filebench benchmarks run simultaneously on the same physical core as hyperthreads. We pick two different types of servers from Filebench- (1) webserver, that emulates web-server I/O activity producing a sequence of open-read-close on multiple files in a directory tree plus a log file append (100 threads are used by default), (2) mailserver, that stores each e-mail in a separate file consisting of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory (16 threads are used by default). The individual benchmarks are chosen based on their CPU-intensive (SPEC2006) and memory- and I/O-intensive (Stream and Filebench) behavior, and are paired such that we maximize the chances of them creating conflicts on a particular microarchitectural unit. As examples, (1) both gobmk and sjeng have numerous repeated accesses to the memory bus, (2) both bzip2 and h264ref have a significant number of integer divisions. The goal of our experiments is to study whether these benchmark pairs create similar levels of recurrent bursts or oscillatory patterns of conflicts that were observed in realistic covert channel implementations (which, if true, could potentially lead to a false alarm). Despite having some regular bursts and conflict cache misses, all of the benchmark pairs are known to not have any covert timing channels. Figure 14



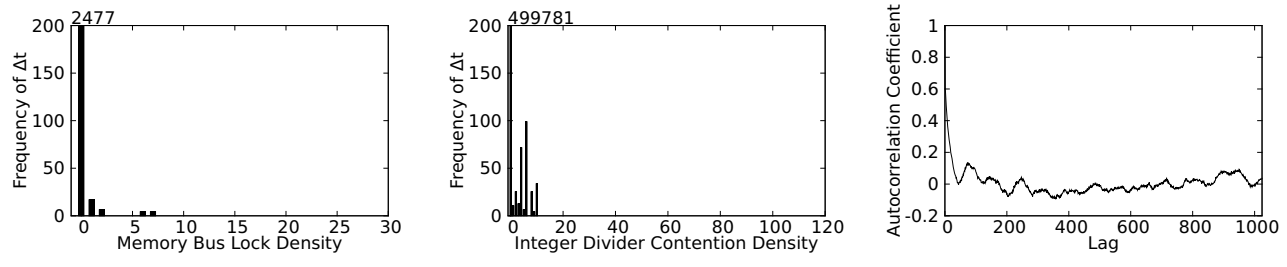
(a) gobmk_sjeng



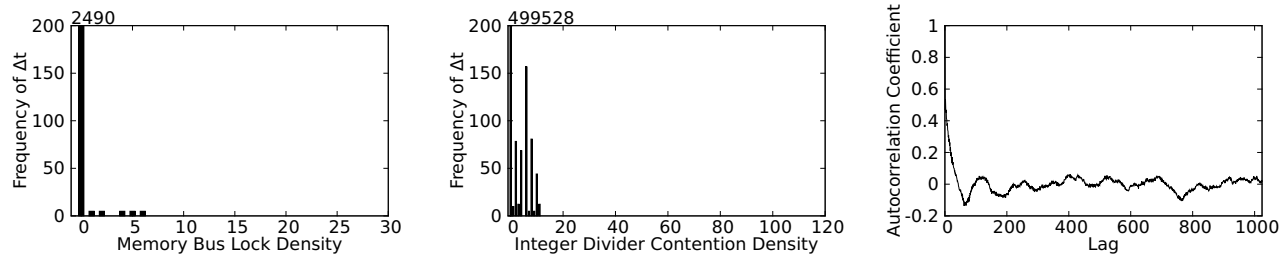
(b) bzip2_h264ref



(c) stream_stream



(d) mailserver_mailserver



(e) webserver_webserver

Figure 14: Event Density Histograms and Autocorrelograms in pair combinations of SPEC2k6, Stream & Filebench

presents a representative subset of our experiments[7]. We observe that most of the benchmark pairs have either zero or random burst patterns for both memory bus lock (first column) and integer division contention (second column) events. The only exception is mailserver pairs, where we observe a second distribution with bursty patterns between histogram bins#5 and #8. Upon further examination, we find that the likelihood ratios for these distributions was less than 0.5 (which is significantly less than the ratios seen in all of our covert timing channel experiments). In almost all of the autocorrelograms (third column), we observe that the autocorrelation coefficients do not exhibit any noticeable periodicity typically expected of cache covert timing channels. The only exception was webserver where we see a very brief period of periodicity between lag values 120 and 180, but becomes non-periodic beyond lag values of 180. Therefore, we did not observe any false alarms. Also, regardless of the "cover" programs that embed the trojan/spy, CC-Hunter is designed to catch the covert transmission phases in the programs that should be already synchronized between the trojan and the spy. Hence, we do not believe that the cover program characteristics could lead to false negatives.

VII. Related Work

The notion of covert channel was first introduced by Lampson et al [29]. Hu et al [3] proposed fuzzing the system clock by randomizing interrupt timer period between 1 ms and 19 ms. Unfortunately, this approach could significantly affect the system's normal bandwidth and performance in the absence of any covert timing channel activity. Recent works have primarily focused on covert information transfer through network channels [30], [31] and mitigation techniques [12], [13], [32]. Among the studies that consider processor-based covert timing channels, Wang and Lee [7] identify two new covert channels using exceptions on speculative load (*ld.s*) instructions and SMT/Multiplier unit. Wu et al. [9] present a high-bandwidth and reliable covert channel attack that is based on the QPI lock mechanism where they demonstrate their results on Amazon's EC2 virtualized environment. Ristenpart et al. [6] present a method of creating a cross-VM covert channel by exploiting the L2 cache, which adopts the Prime+Trigger+Probe [33] to measure the timing difference in accessing two pre-selected cache sets and decipher the covert bit. Xu et al. [10] construct a quantitative study over cross-VM L2 cache covert channels and assess their harm of data exfiltration. Our framework is tested using the examples derived from such prior covert timing channel implementations on shared hardware.

To detect and prevent covert timing channels, Kemmerer et al. [14] proposed a shared matrix methodology to *statically* check whether potential covert communications could happen using shared resources. Wang and Lee [34] propose a covert channel model for an abstract system specification. Unfortunately, such static code-level or abstract model analyses are impractical on every single third-party application executing on a variety of machine configurations in today's computing environments, especially when most of these applications are available in binary-only format.

Side channels are information leakage mechanisms where a certain malware secretly profiles a legitimate application (via differential power, intentional fault injection etc.) to obtain sensitive information. Wang and Lee [16], [35] propose three secure hardware cache designs, Partition-Locking (PL), Random Permutation (RP) and Newcache to defend against cache-based side channel attacks. Kong et al. [15] show how secure software can use the PL cache. Martin et al. [36] propose changes to the infrastructure (timekeeping and performance counters) typically used by side channels such that it becomes difficult for the attackers to derive meaningful clues from architectural events. Demme et al. [37] introduce a metric called Side Channel Vulnerability Factor (SVF) to quantify the level of difficulty for exploiting a particular system to gain side channel information. Many of the above preventative techniques complement CC-Hunter by serving to provide enhanced security to the system.

Demme et al [38] explore simple performance counters for malware analysis. This strategy is not applicable for a number of covert channels because they use specific timing events to modulate hardware resources that may not be measurable through the current performance counter infrastructure. For instance, the integer divider channel should track cycles where one thread waits for another (unsupported by current hardware). Using simple performance counters as alternatives will only lead to a high number of false positives. Also, using machine learning classifiers without considering the time modulation characteristics of covert timing channels could result in false alarms.

VIII. Conclusions

In this paper, we present *CC-Hunter*, a new microarchitecture-level framework to detect the possible presence of covert timing channels on shared processor hardware. Our algorithm works by detecting recurrent burst and oscillation patterns on certain key indicator events associated with the covert timing channels. We test the efficacy of our solution using example covert timing channels on three different types of processor hardware- wires (memory bus/QPI), logic (integer divider) and memory (caches). We conduct sensitivity studies by altering the bandwidth rates, message bit combinations and number of cache sets. Our results show that, at low bandwidths, more frequent analysis (at finer grain windows of observation) may be necessary to improve the probability of detection. Through experiments on I/O, memory, CPU-intensive benchmarks such as Filebench [39], SPEC2006 [40] and Stream [41] that are known to have

[7]Due to space constraints, we are unable to show all of our results.

no covert channels, we show that our framework does not have any false alarms.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557. We thank Dr. Ruby Lee and the anonymous reviewers for their valuable feedback, and Dr. Milos Doroslovacki for his guidance during the initial phase of this work.

REFERENCES

[1] NIST, "National Vulnerability Database," 2013.
[2] J. Gray III, "On introducing noise into the bus-contention channel," in *IEEE Computer Society Symposium on Security and Privacy*, 1993.
[3] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3, 1992.
[4] K. Okamura and Y. Oyama, "Load-based covert channels between xen virtual machines," in *ACM Symposium on Applied Computing*, 2010.
[5] C. Percival, "Cache missing for fun and profit," *BSDCan*, 2005.
[6] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM conference on Computer and communications security*, 2009.
[7] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *IEEE Computer Security Applications Conference*, 2006.
[8] J. C. Wray, "An analysis of covert timing channels," *Journal of Computer Security*, vol. 1, no. 3, 1992.
[9] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *USENIX conference on Security symposium*, 2012.
[10] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *ACM workshop on Cloud computing security workshop*, 2011.
[11] B. Saltaformaggio, D. Xu, and X. Zhang, "Busmonitor: A hypervisor-based solution for memory bus covert channels," *EUROSEC*, 2013.
[12] S. Cabuk, C. E. Brodley, and C. Shields, "Ip covert channel detection," *ACM Transactions on Information and System Security*, vol. 12, no. 4, 2009.
[13] S. Gianvecchio and H. Wang, "Detecting covert timing channels: an entropy-based approach," in *ACM conference on Computer and communications security*, 2007.
[14] R. A. Kemmerer, "Shared resource matrix methodology: An approach to identifying storage and timing channels," *ACM Transactions on Computer Systems*, vol. 1, no. 3, 1983.
[15] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *IEEE Intl. Symp. on High Performance Computer Architecture*, 2009.
[16] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM International symposium on Computer architecture*, 2007.
[17] Department of Defense Standard, *Trusted Computer System Evaluation Criteria*. US Department of Defense, 1983.
[18] H. Okhravi, S. Bak, and S. King, "Design, implementation and evaluation of covert channel attacks," in *International Conference on Technologies for Homeland Security*, 2010.
[19] N. E. Proctor and P. G. Neumann, "Architectural implications of covert channels," in *National Computer Security Conference*, vol. 13, 1992.
[20] Y. Kaneoke and J. Vitek, "Burst and oscillation as disparate neuronal properties," *Journal of neuroscience methods*, vol. 68, no. 2, 1996.
[21] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011*, 2011.
[22] Intel Corporation, "Intel 7500 chipset," *Datasheet*, 2010.
[23] NIST Engineering Statistics Handbook, "Maximum Likelihood," 2013.
[24] G. E. Box, G. M. Jenkins, and G. C. Reinsel, *Time series analysis: forecasting and control*. Wiley, 2011, vol. 734.
[25] G. P. V. Venkataramani, "Low-cost and efficient architectural support for correctness and performance debugging," *Ph.D. Dissertation, Georgia Institute of Technology*, 2009.
[26] HP Labs, "Cacti 5.1," *quid.hpl.hp.com:9081/cacti/*, 2008.
[27] Intel Corporation, "Intel core i7-920 processor," *http://ark.intel.com/Product.aspx?id=37147*, 2010.
[28] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *IEEE International Symposium on Computer Architecture*, 2011.
[29] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, Oct. 1973.
[30] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia, "Model-based covert timing channels: Automated modeling and evasion," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 211–230.
[31] K. Kothari and M. Wright, "Mimic: An active covert channel that evades regularity-based detection," *Comput. Netw.*, vol. 57, no. 3, Feb. 2013.
[32] A. Shabtai, Y. Elovici, and L. Rokach, *A survey of data leakage detection and prevention solutions*. Springer, 2012.
[33] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, no. 2, Jan. 2010.
[34] Z. Wang and R. B. Lee, "New constructive approach to covert channel modeling and channel capacity estimation," in *International Conference on Information Security*, 2005.
[35] Z. Wang and R. Lee, "A novel cache architecture with enhanced performance and security," in *IEEE/ACM International Symposium on Microarchitecture*, 2008.
[36] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *ACM International Symposium on Computer Architecture*, 2012.
[37] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *ACM International Symposium on Computer Architecture*, 2012.
[38] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *IEEE International Symposium on Computer Architecture*, 2013.
[39] File system and Storage Lab, "Filebench," *http://sourceforge.net/apps/mediawiki/filebench*, 2011.
[40] Standard Performance Evaluation Corporation, "Spec 2006 benchmark suite," *www.spec.org*, 2006.
[41] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, 1995.